Learning One-Variable Pattern Languages Very Efficiently on Average, in Parallel, and by Asking Queries ⋆

THOMAS ERLEBACH[1], PETER ROSSMANITH[1], HANS STADTHERR[1], ANGELIKA STEGER[1] AND THOMAS ZEUGMANN[2]

[1] Institut für Informatik, Technische Universität München, 80290 München, Germany, {erlebach|rossmani|stadther|steger}@informatik.tu-muenchen.de
[2] Department of Informatics, Kyushu University, Fukuoka 812-81, Japan, thomas@i.kyushu-u.ac.jp

Abstract. A pattern is a string of constant and variable symbols. The language generated by a pattern π is the set of all strings of constant symbols which can be obtained from π by substituting non-empty strings for variables. We study the learnability of one-variable pattern languages in the limit with respect to the *update time* needed for computing a new single guess and the *expected total learning time* taken until convergence to a correct hypothesis. The results obtained are threefold.
First, we design a *consistent* and *set-driven* learner that, using the concept of descriptive patterns, achieves update time $O(n^2 \log n)$, where n is the size of the input sample. The best previously known algorithm to compute descriptive one-variable patterns requires time $O(n^4 \log n)$ (cf. Angluin [1]). Second, we give a parallel version of this algorithm requiring time $O(\log n)$ and $O(n^3 / \log n)$ processors on an EREW-PRAM. Third, we devise a one-variable pattern learner whose *expected total learning time* is $O(\ell^2 \log \ell)$ provided the sample strings are drawn from the target language according to a probability distribution D with expected string length ℓ. The distribution D must be such that strings of equal length have equal probability, but can be arbitrary otherwise. Thus, we establish the first one-variable pattern learner having an expected total learning time that *provably* differs from the update time by a *constant factor* only.
Finally, we apply the algorithm for finding descriptive one-variable patterns to learn one-variable patterns with a polynomial number of superset queries with respect to the one-variable patterns as *query language*.

1. Introduction

A *pattern* is a string of constant symbols and variable symbols. The language generated by a pattern π is the set of all strings obtained by substituting strings of constants for the variables in π (cf. [1]). Pattern languages and variations thereof have been widely investigated (cf., e.g., [17, 18, 19]). This continuous interest in pattern languages has many reasons, among them the learnability in

⋆ A full version of this paper is available as technical report (cf. [4]).

the limit of all pattern languages from text (cf. [1, 2]). Moreover, the learnability of pattern languages is very interesting with respect to potential applications (cf., e.g., [19]). Given this, *efficiency* becomes a central issue. However, defining an appropriate measure of efficiency for learning in the limit is a difficult problem (cf. [16]). Various authors studied the efficiency of learning in terms of the *update time* needed for computing a new single guess. But what really counts in applications is the overall time needed by a learner until convergence, i.e., the *total learning time*. One can show that the total learning time is *unbounded* in the worst-case. Thus, we study the *expected* total learning time. For the purpose of motivation we shortly summarize what has been known in this regard.

The pattern languages can be learned by outputting *descriptive patterns* as hypotheses (cf. [1]). The resulting learning algorithm is *consistent* and *set-driven*. A learner is set-driven iff its output depends only on the range of its input (cf., e.g., [20, 22]). In general, consistency and set-drivenness considerably limit the learning capabilities (cf., e.g., [14, 22]). But no polynomial time algorithm computing descriptive patterns is known, and hence already the update time is practically infeasible. Moreover, finding a descriptive pattern of *maximum* possible length is $\mathcal{NP}$-complete (cf. [1]). Thus, it is unlikely that there is a polynomial-time algorithm computing descriptive patterns of *maximum* possible length.

It is therefore natural to ask whether efficient pattern learners can benefit from the concept of descriptive patterns. Special cases, e.g., *regular* patterns, *non-cross* patterns, and unions of at most k regular pattern languages (k *a priori* fixed) have been studied. In all these cases, descriptive patterns are polynomial-time computable (cf., e.g., [19]), and thus these learners achieve polynomial update time but nothing is known about their expected total learning time. Another restriction is obtained by *a priori* bounding the number k of different variables occurring in a pattern (k-variable patterns) but it is open if there are polynomial-time algorithms computing descriptive k-variable patterns for any fixed $k > 1$ (cf. [9, 12]). On the other hand, k-variable patterns are *PAC*–learnable with respect to unions of k-variable patterns as hypothesis space (cf. [11]).

Lange and Wiehagen [13] provided a learner *LWA* for all pattern languages that may output *inconsistent* guesses. The *LWA* achieves *polynomial update time*. It is set-driven (cf. [21]), and even *iterative*, thus beating Angluin's [1] algorithm with respect to its space complexity. For the *LWA*, the expected total learning time is exponential in the number of different variables occurring in the target pattern (cf. [21]). Moreover, the point of convergence for the *LWA* definitely depends on the appearance of sufficiently many *shortest* strings of the target language, while for the other algorithms mentioned above at least the corresponding correctness proofs depend on it. Thus, the following problem arises naturally.

Does there exist an efficient pattern language learner benefiting from the concept of descriptive patterns thereby not depending on the presentation of sufficiently many shortest strings from the target language, and still achieving an expected total learning time polynomially bounded in the expected string length?

We provide a complete affirmative answer by studying the special case of one-variable patterns. We believe this case to be a natural choice, since it is nontrivial (there may be exponentially many consistent patterns for a given sample), and since it has been the first case for which a polynomial time algorithm computing descriptive patterns has been known (cf. [1]). Angluin's [1] algorithm for finding descriptive patterns runs in time $O(n^4 \log n)$ for inputs of size n and it always outputs descriptive patterns of *maximum* possible length. She was aware of possible improvements of the running time only for certain special cases, but hoped that further study would provide insight for a uniform improvement.

We present such an improvement, i.e., an algorithm computing descriptive one-variable patterns in $O(n^2 \log n)$ steps. A key idea to achieve this goal is giving up necessarily finding descriptive patterns of *maximum* possible length. Note that all results concerning the difficulty of finding descriptive patterns depend on the additional requirement to compute ones having maximum possible length (cf., e.g., [1, 12]). Thus, our result may at least support the conjecture that more efficient learners may arise when one is definitely not trying to find descriptive patterns of *maximum* possible length but just descriptive ones, instead.

Moreover, our algorithm can be also efficiently parallelized, using $O(\log n)$ time and $O(n^3/\log n)$ processors on an EREW-PRAM. Previously, no efficient parallel algorithm for learning one-variable pattern languages was known.

Our main result is a version of the sequential algorithm still learning all one-variable pattern languages that has *expected total learning time* $O(\ell^2 \log \ell)$ if the sample strings are drawn from the target language according to a probability distribution D having expected string length ℓ. D can be arbitrary except that strings of equal length have equal probability. In particular, all shortest strings may have probability 0. Note that the expected total learning time differs only by a *constant factor* from the time needed to update actual hypotheses. On the other hand, we could only prove that $O(\log \ell)$ many examples are sufficient to achieve convergence.

Finally, we deal with *active* learning. Now the learner gains information about the target object by *asking queries* to an oracle. We show how the algorithm for descriptive one-variable patterns can be used for learning one-variable patterns by asking polynomially many superset queries. Another algorithm learning all pattern languages with a polynomial number of superset queries is known, but it uses a much more *powerful* query language, i.e., patterns with *more than one variable* even when the target pattern is a one-variable pattern (cf. [3]).

1.1. The Pattern Languages and Inductive Inference

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of all natural numbers, and let $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. For all real numbers x we define $\lfloor x \rfloor$ to be the greatest integer less than or equal to x. Let $\mathcal{A} = \{0, 1, \ldots\}$ be any finite alphabet containing at least two elements. $\mathcal{A}^*$ denotes the free monoid over $\mathcal{A}$. By $\mathcal{A}^+$ we denote the set of all finite non-null strings over $\mathcal{A}$, i.e., $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$, where ε denotes the empty string. Let $X = \{x_i \mid i \in \mathbb{N}\}$ be an infinite set of variables with $\mathcal{A} \cap X = \emptyset$. *Patterns* are strings from $(\mathcal{A} \cup X)^+$. The length of a string $s \in \mathcal{A}^*$ and of a pattern π is denoted by $|s|$ and $|\pi|$, respectively. By *Pat* we denote the set of all patterns.

Let $\pi \in Pat$, $1 \le i \le |\pi|$; by $\pi(i)$ we denote the i-th symbol in π, e.g., $0x_0111(2) = x_0$. If $\pi(i) \in \mathcal{A}$, then $\pi(i)$ is called a *constant*; otherwise $\pi(i) \in X$, i.e., $\pi(i)$ is a *variable*. We use $s(i)$, $i = 1, \dots, |s|$, to denote the i-th symbol in $s \in \mathcal{A}^+$. By $\#\mathrm{var}(\pi)$ we denote the number of different variables occurring in π, and $\#_{x_i}(\pi)$ denotes the number of occurrences of variable x_i in π. If $\#\mathrm{var}(\pi) = k$, then π is a *k-variable pattern*. By Pat_k we denote the set of all *k-variable patterns*. In the case $k = 1$ we denote the variable occurring by x.

Let $\pi \in Pat_k$, and $u_0, \dots, u_{k-1} \in \mathcal{A}^+$. $\pi[u_0/x_0, \dots, u_{k-1}/x_{k-1}]$ denotes the string $s \in \mathcal{A}^+$ obtained by substituting u_j for each occurrence of x_j in π. The tuple $(u_0, \dots, u_{k-1})$ is called *substitution*. For every pattern $\pi \in Pat_k$ we define the *language generated* by π by $L(\pi) = \{\pi[u_0/x_0, \dots, u_{k-1}/x_{k-1}] \mid u_0, \dots, u_{k-1} \in \mathcal{A}^+\}$[3]. By PAT_k we denote the set of all *k-variable pattern languages*. $PAT = \bigcup_{k \in \mathbb{N}} PAT_k$ denotes the set of all pattern languages over $\mathcal{A}$.

Note that several problems are decidable or even efficiently solvable for PAT_1 but undecidable or $\mathcal{NP}$–complete for PAT. For example, for general pattern languages the word problem is $\mathcal{NP}$–complete (cf. [1]) and the inclusion problem is undecidable (cf. [10]), but both problems are decidable in linear time for one-variable pattern languages. On the other hand, PAT_1 is still incomparable to the regular and context free languages.

A finite set $S = \{s_0, s_1, \dots, s_r\} \subseteq \mathcal{A}^+$ is called a *sample*. A pattern π is *consistent* with a sample S if $S \subseteq L(\pi)$. A (one-variable) pattern π is called *descriptive* for S if it is consistent with S and there is no other consistent (one-variable) pattern τ such that $L(\tau) \subset L(\pi)$.

Next, we define the relevant learning models. We start with inductive inference of languages from text (cf., e.g., [15, 22]). Let L be a language; every infinite sequence $t = (s_j)_{j \in \mathbb{N}}$ of strings with $\mathrm{range}(t) = \{s_j \mid j \in \mathbb{N}\} = L$ is said to be a *text* for L. $Text(L)$ denotes the set of all texts for L. Let t be a text, and $r \in \mathbb{N}$. We set $t_r = s_0, \dots, s_r$, and call t_r the initial segment of t of length $r + 1$. By t_r^+ we denote the range of t_r, i.e., $t_r^+ = \{s_j \mid 0 \le i \le r\}$. We define an *inductive inference machine* (abbr. IIM) to be an algorithmic device taking as its inputs initial segments of a text t, and outputting on every input a pattern as hypothesis (cf. [7]).

Definition 1. *PAT is called learnable in the limit from text iff there is an IIM M such that for every $L \in PAT$ and every $t \in Text(L)$,*

(1) *for all $r \in \mathbb{N}$, $M(t_r)$ is defined,*
(2) *there is a $\pi \in Pat$ such that $L(\pi) = L$ and for almost all $r \in \mathbb{N}$, $M(t_r) = \pi$.*

The learnability of the one-variable pattern languages is defined analogously by replacing PAT and Pat by PAT_1 and Pat_1, respectively.

When dealing with set-driven learners, it is often technically advantageous to describe them in dependence of the relevant set t_r^+, i.e., a *sample*, obtained as input. Let $S = \{s_0, s_1, \dots, s_r\}$; we set $n = \sum_{j=0}^{r} |s_j|$, and refer to n as the *size* of sample S.

---

[3] We study *non-erasing* substitutions. *Erasing* substitutions have been also considered, (variables may be replaced by ε), leading to a different language class (cf. [5]).

PAT as well as PAT_1 constitute an *indexable class* $\mathcal{L}$ of uniformly recursive languages, i.e., there are an effective enumeration $(L_j)_{j\in\mathbb{N}}$ of $\mathcal{L}$ and a recursive function f such that for all $j \in \mathbb{N}$ and all $s \in \mathcal{A}^*$ we have $f(j,s) = 1$ if $s \in L_j$, and $f(j,s) = 0$ otherwise.

Except in Section 3, where we use the PRAM-model, we assume the same model of computation and representation of patterns as in [1]. Next we define the update time and the total learning time. Let M be any IIM. For every $L \in PAT$ and $t \in Text(L)$, let $Conv(M,t) =$ the least number m such that for all $r \geq m$, $M(t_r) = M(t_m)$ denote the *stage of convergence* of M on t. By $T_M(t_r)$ we denote the time to compute $M(t_r)$, and we call $T_M(t_r)$ the *update time* of M. The *total learning time* taken by the IIM M on successive input t is defined as $TT(M,t) = \sum_{r=0}^{Conv(M,t)} T_M(t_r)$.

Finally, we define *learning via queries*. The objects to be learned are the elements of an indexable class $\mathcal{L}$. We assume an indexable class $\mathcal{H}$ as well as a fixed effective enumeration $(h_j)_{j\in\mathbb{N}}$ of it as hypothesis space for $\mathcal{L}$. Clearly, $\mathcal{H}$ must *comprise* $\mathcal{L}$. A hypothesis h describes a target language L iff $L = h$. The source of information about the target L are queries to an *oracle*. We distinguish *membership, equivalence, subset, superset,* and *disjointness* queries (cf. [3]). Input to a membership query is a string s, and the output is *yes* if $s \in L$ and *no* otherwise. For the other queries, the input is an index j and the output is *yes* if $L = h_j$ (equivalence query), $h_j \subseteq L$ (subset query), $L \subseteq h_j$ (superset query), and $L \cap h_j = \emptyset$ (disjointness query), and *no* otherwise. If the reply is no, a *counterexample* is returned, too, i.e., a string $s \in L \triangle h_j$ (the symmetric difference of L and h_j), $s \in h_j \setminus L$, $s \in L \setminus h_j$, and $s \in L \cap h_j$, respectively. We always assume that all queries are *answered truthfully*.

Definition 2 (Angluin [3]). *Let $\mathcal{L}$ be any indexable class and let $\mathcal{H}$ be a hypothesis space for it. A learning algorithm exactly identifies a target $L \in \mathcal{L}$ with respect to $\mathcal{H}$ with access to a certain type of queries if it always halts and outputs an index j such that $L = h_j$.*

Note that the learner is allowed only *one hypothesis* which must be correct. The *complexity* of a query learner is measured by the number of queries to be asked in the worst-case.

2. An Improved Sequential Algorithm

We present an algorithm computing a descriptive pattern $\pi \in Pat_1$ for a sample $S = \{s_0, s_1, \ldots, s_{r-1}\} \subseteq \mathcal{A}^+$ as input. Without loss of generality, we assume that s_0 is the shortest string in S. Our algorithm runs in time $O(n\,|s_0|\log|s_0|)$ and is simpler and much faster than Angluin's [1] algorithm, which needs time $O(n^2|s_0|^2\log|s_0|)$. Angluin's [1] algorithm computes explicitly a representation of the set of all consistent one-variable patterns for S and outputs a descriptive pattern of *maximum* possible length. We avoid to find a descriptive pattern of *maximum* possible length and can thus work with a *polynomial-size* subset of all consistent patterns. Next, we review and establish basic properties needed later.

Lemma 1 (Angluin [1]). *Let $\tau \in Pat$, and let $\pi \in Pat_1$. Then $L(\tau) \subseteq L(\pi)$ if and only if τ can be obtained from π by substituting a pattern $\varrho \in Pat$ for x.*

For a pattern π to be consistent with S, there must be strings $\alpha_0, \ldots, \alpha_{r-1} \in \mathcal{A}^+$ such that s_i can be obtained from π by substituting α_i for x, for all $0 \le i \le r-1$. Given a consistent pattern π, the set $\{\alpha_0, \ldots, \alpha_{r-1}\}$ is denoted by $\alpha(S, \pi)$. Moreover, a sample S is called *prefix-free* if $|S| > 1$ and no string in S is a prefix of all other strings in S. Note that the distinction between prefix-free samples and non-prefix free samples does well pay off.

Lemma 2. *If S is prefix-free then there exists a descriptive pattern $\pi \in Pat_1$ for S such that at least two strings in $\alpha(S, \pi)$ start with a different symbol.*

Proof. Let u denote the longest common prefix of all strings in S. The pattern ux is consistent with S because u is shorter than every string in S, since S is prefix-free. Consequently, there exists a descriptive pattern $\pi \in Pat_1$ for S with $L(\pi) \subseteq L(ux)$. Now, by Lemma 1 we know that there exists a pattern $\varrho \in Pat_1$ such that $\pi = ux[\varrho/x]$. Since u is a longest common prefix of all strings in S, we can conclude $\varrho(1) \notin \mathcal{A}$. Hence, $\varrho = x\tau$ for some $\tau \in Pat_1 \cup \{\varepsilon\}$, and at least two strings in $\alpha(S, ux\tau)$ must start with a different symbol. $\square$

Let $Cons(S) = \{\pi \mid \pi \in Pat_1,\ S \subseteq L(\pi),\ \exists i, j[i \neq j,\ \alpha_i,\ \alpha_j \in \alpha(S, \pi),\ \alpha_i(1) \neq \alpha_j(1)]\}$. $Cons(S)$ is a subset of all consistent patterns for S, and $Cons(S) = \emptyset$ if S is not prefix-free.

Lemma 3. *Let S be any prefix-free sample. Then $Cons(S) \neq \emptyset$, and every pattern $\pi \in Cons(S)$ of maximum length is descriptive for S.*

Proof. Let S be prefix-free. According to Lemma 2 there exists a descriptive pattern for S belonging to $Cons(S)$; thus $Cons(S) \neq \emptyset$. Now, suppose there is a pattern $\pi \in Cons(S)$ of maximum length which is not descriptive for S. Thus, $S \subseteq L(\pi)$, and, moreover, there exists a pattern $\tau \in Pat_1$ such that $S \subseteq L(\tau)$ as well as $L(\tau) \subset L(\pi)$. Hence, by Lemma 1 we know that τ can be obtained from π by substituting a pattern ϱ for x. Since at least two strings in $\alpha(S, \pi)$ start with a different symbol, we immediately get $\varrho(1) \in X$. Moreover, at least two strings in $\alpha(S, \tau)$ must also start with a different symbol. Hence $\tau \in Cons(S)$ and $\varrho = x\nu$ for some pattern ν. Note that $|\nu| \ge 1$, since otherwise $\tau = \pi[\varrho/x] = \pi$ contradicting $L(\tau) \subset L(\pi)$. Finally, by $|\nu| \ge 1$, we may conclude $|\tau| > |\pi|$, a contradiction to π having maximum length. Thus, no such pattern τ can exist, and hence, π is descriptive. $\square$

Next, we explain how to handle non-prefix-free samples. The algorithm checks whether the input sample consists of a single string s. If this happens, it outputs s and terminates. Otherwise, it tests whether s_0 is a prefix of all other strings $s_1, \ldots, s_{r-1}$. If this is the case, it outputs $ux \in Pat_1$, where u is the prefix of s_0 of length $|s_0| - 1$, and terminates. Clearly, $S \subseteq L(ux)$. Suppose there is a pattern τ such that $S \subseteq L(\tau)$, and $L(\tau) \subset L(ux)$. Then Lemma 1 applies, i.e., there is a ϱ such that $\tau = ux[\varrho/x]$. But this implies $\varrho = x$, since otherwise $|\tau| > |s_0|$, and thus, $S \not\subseteq L(\tau)$. Consequently, ux is descriptive.

Otherwise, $|S| \ge 2$ and s_0 is not a prefix of all other strings in S. Thus, by Lemma 3 it suffices to find and to output a longest pattern in $Cons(S)$.

The improved algorithm for *prefix-free* samples is based on the fact that $|Cons(S)|$ is bounded by a small polynomial. Let $k, l \in \mathbb{N}$, $k > 0$; we call patterns π with $\#_x(\pi) = k$ and l occurrences of constants (k,l)-patterns. A (k,l)-pattern has length $k+l$. Every pattern $\pi \in \mathit{Cons}(S)$ satisfies $|\pi| \le |s_0|$. Thus, there can only be a (k,l)-pattern in $\mathit{Cons}(S)$ if there is an $m_0 \in \mathbb{N}^+$ satisfying $|s_0| = km_0 + l$. Here m_0 refers to the length of the string substituted for the occurrences of x in the relevant (k,l)-pattern to obtain s_0. Therefore, there are at most $\lfloor |s_0|/k \rfloor$ possible values of l for a fixed value of k. Hence, the number of possible (k,l)-pairs for which (k,l)-patterns in $\mathit{Cons}(S)$ can exist is bounded by $\sum_{k=1}^{|s_0|} \left\lfloor \frac{|s_0|}{k} \right\rfloor = O(|s_0| \cdot \log |s_0|)$.

The algorithm considers all possible (k,l)-pairs in turn. We describe the algorithm for one specific (k,l)-pair. If there is a (k,l)-pattern $\pi \in \mathit{Cons}(S)$, the lengths m_i of the strings $\alpha_i \in \alpha(S,\pi)$ must satisfy $m_i = (|s_i| - l)/k$. α_i is the substring of s_i of length m_i starting at the first position where the input strings differ. If $(|s_i| - l)/k \notin \mathbb{N}$ for some i, then there is no consistent (k,l)-pattern and no further computation is performed for this (k,l)-pair. The following lemma shows that the (k,l)-pattern in $\mathit{Cons}(S)$ is unique, if it exists at all.

Lemma 4. *Let $S = \{s_0, \ldots, s_{r-1}\}$ be any prefix-free sample. For every given (k,l)-pair, there is at most one (k,l)-pattern in $\mathit{Cons}(S)$.*

The proof of Lemma 4 directly yields Algorithm 1 below. It either returns the unique (k,l)-pattern $\pi \in \mathit{Cons}(S)$ or *NIL* if there is no (k,l)-pattern in $\mathit{Cons}(S)$. We assume a subprocedure taking as input a sample S, and returning the longest common prefix u.

***Algorithm* 1.** On input (k,l), $S = \{s_0, \ldots, s_{r-1}\}$, and u do the following:

$\pi \leftarrow u$; $b \leftarrow 0$; $c \leftarrow |u|$;

while $b + c < k + l$ and $b \le k$ and $c \le l$ **do**

  if $s_0(bm_0 + c + 1) = s_i(bm_i + c + 1)$ for all $1 \le i \le r - 1$

    then $\pi \leftarrow \pi s_0(bm_0 + c + 1)$; $c \leftarrow c + 1$

    else $\pi \leftarrow \pi x$; $b \leftarrow b + 1$ **fi** **od**;

if $b = k$ and $c = l$ and $S \subseteq L(\pi)$ **then return** π **else return** *NIL* **fi**

Note that minor modifications of Algorithm 1 perform the consistency test $S \subseteq L(\pi)$ even while π is constructed. Putting Lemma 4 and the fact that there are $O(|s_0| \cdot \log |s_0|)$ many possible (k,l)-pairs together, we directly obtain:

Lemma 5. $|\mathit{Cons}(S)| = O(|s_0| \log |s_0|)$ *for every prefix-free sample* $S = \{s_0, \ldots, s_{r-1}\}$.

Using Algorithm 1 as a subroutine, Algorithm 2 below for finding a descriptive pattern for a prefix-free sample S follows the strategy exemplified above. Thus, it simply computes all patterns in $\mathit{Cons}(S)$ and outputs one with maximum length. For inputs of size n the overall complexity of the algorithm is $O(n\,|s_0| \log |s_0|) = O(n^2 \log n)$, since at most $O(|s_0| \log |s_0|)$ many tests must be performed, which have time complexity $O(n)$ each.

***Algorithm* 2.** On input $S = \{s_0, \ldots, s_{r-1}\}$ do the following:

$P \leftarrow \emptyset$; **for** $k = 1, \ldots, |s_0|$ and $m_0 = 1, \ldots, \left\lfloor \frac{|s_0|}{k} \right\rfloor$ **do**

    if there is a $(k, |s_0| - km_0)$-pattern $\pi \in \mathit{Cons}(S)$ **then** $P \leftarrow P \cup \{\pi\}$ **fi** **od**;

Output a maximum-length pattern $\pi \in P$.

Note that the number of (k,l)-pairs to be processed is often smaller than $O(|s_0| \log |s_0|)$, since the condition $(|s_i| - l)/k \in \mathbb{N}$ for all i restricts the possible values of k if not all strings are of equal length. It is also advantageous to process the (k,l)-pairs in order of non-increasing $k+l$. Then the algorithm can terminate as soon as it finds the first consistent pattern. However, the worst-case complexity is not improved, if the descriptive pattern is x.

Finally, we summarize the main result obtained by the following theorem.

Theorem 1. *Using Algorithm 2 as a subroutine, PAT_1 can be learned in the limit by a set-driven and consistent IIM having update time $O(n^2 \log n)$ on input samples of size n.*

3. An Efficient Parallel Algorithm

Whereas the RAM model has been generally accepted as the most suitable model for developing and analyzing sequential algorithms, such a consensus has not yet been reached in the area of parallel computing. The PRAM model introduced in [6], is usually considered an acceptable compromise. A PRAM consists of a number of processors, each of which has its own local memory and can execute its local program, and all of which can communicate by exchanging data through a shared memory. Variants of the PRAM model differ in the constraints on simultaneous accesses to the same memory location by different processors. The CREW-PRAM allows concurrent read accesses but no concurrent write accesses. For ease of presentation, we describe our algorithm for the CREW-PRAM model. The algorithm can be modified to run on an EREW-PRAM, however, by the use of standard techniques.

No parallel algorithm for computing descriptive one-variable patterns has been known previously. Algorithm 2 can be efficiently parallelized by using well-known techniques including prefix-sums, tree-contraction, and list-ranking as subroutines (cf. [8]). A parallel algorithm can handle non-prefix-free samples S in the same way as Algorithm 2. Checking S to be singleton or s_0 to be a prefix of all other strings requires time $O(\log n)$ using $O(n/\log n)$ processors. Thus, we may assume that the input sample $S = \{s_0, \ldots, s_{r-1}\}$ is prefix-free. Additionally, we assume the prefix-test has returned the first position d where the input strings differ and an index t, $1 \leq t \leq r-1$, such that $s_0(d) \neq s_t(d)$.

A parallel algorithm can handle all $O(|s_0| \log |s_0|)$ possible (k,l)-pairs in parallel. For each (k,l)-pair, our algorithm computes a unique candidate π for the (k,l)-pattern in $Cons(S)$, if it exists, and checks whether $S \subseteq L(\pi)$. Again, it suffices to output any obtained pattern having maximum length. Next, we show how to efficiently parallelize these two steps.

For a given (k,l)-pair, the algorithm uses only the strings s_0 and s_t for calculating the unique candidate π for the (k,l)-pattern in $Cons(S)$. This reduces the processor requirements, and a modification of Lemma 4 shows the candidate pattern to remain unique.

Position j_t in s_t is said to be b-corresponding to position j_0 in s_0 if $j_t = j_0 + b(m_t - m_0)$, $0 \leq b \leq k$. The meaning of b-corresponding positions is as follows.

Suppose there is a consistent (k,l)-pattern π for $\{s_0,\, s_t\}$ such that position j_0 in s_0 corresponds to $\pi(i) \in \mathcal{A}$ for some i, $1 \le i \le |\pi|$, and that b occurrences of x are to the left of $\pi(i)$. Then $\pi(i)$ corresponds to position $j_t = j_0 + b(m_t - m_0)$ in s_t.

For computing the candidate pattern from s_0 and s_t, the algorithm calculates the entries of an array EQUAL$[j,b]$ of Boolean values first, where j ranges from 1 to $|s_0|$ and b from 0 to k. EQUAL$[j,b]$ is true iff the symbol in position j in s_0 is the same as the symbol in its b-corresponding position in s_t. Thus, the array is defined as follows: EQUAL$[j,b]$ = *true* iff $s_0(j) = s_t(j + b(m_t - m_0))$. The array EQUAL has $O(k|s_0|)$ entries each of which can be calculated in constant time. Thus, using $O(k|s_0|/\log n)$ processors, EQUAL can be computed in time $O(\log n)$. Moreover, a directed graph G that is a forest of binary in-trees, can be built from EQUAL, and the candidate pattern can be calculated from G using tree-contraction, prefix-sums and list-ranking. The details are omitted due to space restrictions. Thus, we can prove:

Lemma 6. *Let $S = \{s_0,\, \dots,\, s_{r-1}\}$ be a sample, and n its size. Given the array* EQUAL*, the unique candidate π for the (k,l)-pattern in Cons(S), or NIL, if no such pattern exists, can be computed on an EREW-PRAM in time $O(\log n)$ using $O(k|s_0|/\log n)$ processors.*

Now, the algorithm has either discovered that no (k,l)-pattern exists, or it has obtained a candidate (k,l)-pattern π. In the latter case, it has to test whether π is consistent with S.

Lemma 7. *Given a candidate pattern π, the consistency of π with any sample S of size n can be checked on in time $O(\log n)$ using $O(n/\log n)$ processors on a CREW-PRAM.*

Putting it all together, we obtain the following theorem.

Theorem 2. *There exists a parallel algorithm computing descriptive one-variable patterns in time $O(\log n)$ using $O(|s_0|\max\{|s_0|^2, n\log|s_0|\}/\log n) = O(n^3/\log n)$ processors on a CREW-PRAM for samples $S = \{s_0,\, \dots,\, s_{r-1}\}$ of size n.*

Note that the product of the time and the number of processors of our algorithm is the same as the time spent by the improved sequential algorithm above whenever $|s_0|^2 = O(n\log|s_0|)$. If $|s_0|$ is larger, the product exceeds the time of the sequential algorithm by a factor less than $O(|s_0|^2/n) = O(|s_0|)$.

4. Analyzing the Expected Total Learning Time

Now, we are dealing with the major result of the present paper, i.e., with the expected total learning time of our sequential learner. Let π be the target pattern. The total learning time of any algorithm trying to infer π is *unbounded* in the worst case, since there are infinitely many strings in $L(\pi)$ that can mislead it. However, in the *best case* two examples, i.e., the two *shortest* strings $\pi[0/x]$ and $\pi[1/x]$, always suffice for a learner outputting descriptive patterns as guesses. Hence, we assume that the strings presented to the algorithm are drawn from $L(\pi)$ according to a certain probability distribution D and compute the *expected* *total learning time* of our algorithm. The distribution D must satisfy two criteria: any two strings in $L(\pi)$ of equal length must have equal probability, and the expected string length ℓ must be finite. We refer to such distributions as *proper* probability distributions.

We design an Algorithm 1LA inferring a pattern $\pi \in Pat_1$ with expected total learning time $O(\ell^2 \log \ell)$. It is advantageous *not* to calculate a descriptive pattern each time a new string is read. Instead, Algorithm 1LA reads a certain number of strings before it starts to perform any computations at all. It waits until the length of a sample string is smaller than the number of sample strings read so far and until at least two *different* sample strings have been read. During these first two phases, it outputs s_1, the first sample string, as its guess if all sample strings read so far are the same, and x otherwise. If π is a constant pattern, i.e., $|L(\pi)| = 1$, the correct hypothesis is always output and the algorithm never reaches the third phase. Otherwise, the algorithm uses a modified version of Algorithm 2 to calculate a set P' of candidate patterns when it enters Phase 3. More precisely, it does not calculate the whole set P' at once. Instead, it uses the function *first_cand* once to obtain a longest pattern in P', and the function *next_cand* repeatedly to obtain the remaining patterns of P' in order of non-increasing length. This substantially reduces the memory requirements.

The pattern τ obtained from calling *first_cand* is used as the *current* candidate. Each new string s is then compared to τ. If $s \in L(\tau)$, τ is output. Otherwise, *next_cand* is called to obtain a new candidate τ'. Now, τ' is the current candidate and output, independently of $s \in L(\tau')$. If the longest common prefix of all sample strings including s is shorter than that of all sample strings excluding s, however, *first_cand* is called again and a new list of candidate patterns is considered. Thus, Algorithm 1LA may output *inconsistent* hypotheses.

Algorithm 1LA is shown in Figure 1. Let $S = \{s, s'\}$, and let $P' = P'(S)$ be defined as follows. If $s = vc$ for some $c \in \mathcal{A}$ and $s' = sw$ for some string w, then $P' = \{vx\}$. Otherwise, denote by u the longest common prefix of s and s', and let P' be the set of patterns computed by Algorithms 1 and 2 above if we omit the consistency check. Hence, $P' \supseteq Cons(S)$, where $Cons(S)$ is defined as in Section 2. P' necessarily contains the pattern π if $s, s' \in L(\pi)$ and if the longest common prefix of s and s' is the same as the longest constant prefix of π. Assuming $P' = \{\pi_1, \ldots, \pi_t\}$, $|\pi_i| \geq |\pi_{i+1}|$ for $1 \leq i < t$, *first_cand*(s, s') returns π_1, and *next_cand*(s, s', π_i) returns π_{i+1}. Since we omit the consistency checks, a call to *first_cand* and all subsequent calls to *next_cand* until either the correct pattern is found or the prefix changes can be performed in time $O(|s|^2 \log |s|)$.

We will show that Algorithm 1LA correctly infers one-variable pattern languages from text in the limit, and that it correctly infers one-variable pattern languages from text with probability 1 if the sample strings are drawn from $L(\pi)$ according to a proper probability distribution.[4]

Theorem 3. *Let π be an arbitrary one-variable pattern. Algorithm 1LA correctly infers π from text in the limit.*

[4] Note that learning a language L in the limit and learning L from strings that are drawn from L according to a proper probability distribution are not the same.

```
{ Phase 1 }
r ← 0;
repeat r ← r + 1; read string s_r;
   if s_1 = s_2 = ··· = s_r then output hypothesis s_1 else output hypothesis x fi
until |s_r| < r;

{ Phase 2 }
while s_1 = s_2 = ··· = s_r do r ← r + 1; read string s_r;
   if s_r = s_1 then output hypothesis s_1 else output hypothesis x fi
od;

{ Phase 3 }
s ← a shortest string in {s_1, s_2, ..., s_r};
u ← maximum length common prefix of {s_1, s_2, ..., s_r};
if u = s then s' ← a string in {s_1, s_2, ..., s_r} that is longer than s
else s' ← a string in {s_1, s_2, ..., s_r} that differs from s in position |u| + 1 fi
τ ← first_cand(s, s');
forever do
   read string s'';
   if u is not a prefix of s'' then
      u ← maximum length common prefix of s and s'';
      s' ← s'';
      τ ← first_cand(s, s')
   else if s'' ∉ L(τ) then τ ← next_cand(s, s', τ) fi;
   output hypothesis τ;
od
```

Fig. 1. Algorithm 1LA

Theorem 4. *Let $\pi \in Pat_1$. If sample strings are drawn from $L(\pi)$ according to a proper probability distribution, Algorithm 1LA learns π with probability* 1.

Proof. If $\pi \in \mathcal{A}^+$ then Algorithm 1LA outputs π after reading the first string and converges. Otherwise, let $\pi = ux\mu$, where $u \in \mathcal{A}^*$ is a string of $d-1$ constant symbols and $\mu \in Pat_1 \cup \{\varepsilon\}$. After Algorithm 1LA has read two strings that differ in position d, pattern π will be one of the candidates in the set P' implicitly maintained by the algorithm. As each string s_r, $r > 1$, satisfies $s_1(d) \neq s_r(d)$ with probability $(|\mathcal{A}|-1)/|\mathcal{A}| \geq 1/2$, this event must happen with probability 1. After that, as long as the current candidate τ differs from π, the probability that the next string read does not belong to $L(\tau)$ is at least $1/2$ (cf. Lemma 8 below). Hence, all candidate patterns will be discarded with probability 1 until π is the current candidate and is output. After that, the algorithm converges. □

Lemma 8. *Let $\pi = ux\mu$ be a one-variable pattern with constant prefix u, and $\mu \in Pat_1 \cup \{\varepsilon\}$. Let s_0, $s_1 \in L(\pi)$ be arbitrary such that $s_0(|u|+1) \neq s_1(|u|+1)$. Let $\tau \neq \pi$ be a pattern from $P'(\{s_0, s_1\})$ with $|\tau| \geq |\pi|$. Then τ fails to generate a string s drawn from $L(\pi)$ according to a proper probability distribution with probability at least $(|\mathcal{A}|-1)/|\mathcal{A}|$.*

Now, we analyze the total expected learning time of Algorithm 1LA. Obviously, the total expected learning time is $O(\ell)$ if the target pattern $\pi \in \mathcal{A}^+$. Hence, we assume in the following that π contains at least one occurrence of x.

Next, we recall the definition of the median, and establish a basic property of it that is used later. By $E(R)$ we denote the *expectation* of a random variable R.

Definition 3. *Let R be any random variable with* $\mathrm{range}(R) \subseteq \mathbb{N}$. *The median of R is the number $\mu \in \mathrm{range}(R)$ such that $Pr(R < \mu) \leq \frac{1}{2}$ and $Pr(R > \mu) < \frac{1}{2}$.*

Proposition 1. *Let R be a random variable with* $\mathrm{range}(R) \subseteq \mathbb{N}^+$. *Then its median μ satisfies $\mu \leq 2E(R)$.*

Lemma 9. *Let D be any proper probability distribution, let L be the random variable taking as values the length of a string drawn from $L(\pi)$ with respect to D, and let μ be the median of L and let ℓ be its expectation. Then, the expected number of steps performed by Algorithm 1LA during Phase 1 is $O(\mu\ell)$.*

Proof. Let L_i be the random variable whose value is the length of the i-th string read by the algorithm. Clearly, the distribution of L_i is the same as that of L. Let R be the random variable whose value is the number of strings Algorithm 1LA reads in Phase 1. Let $L_\Sigma := L_1 + \cdots + L_R$ be the number of symbols read by Algorithm 1LA during Phase 1. Let W_1 be the random variable whose value is the time spent by Algorithm 1LA in Phase 1. Obviously, $W_1 = O(L_\Sigma)$.

Claim 1. *For $i < r$, we have:* $$E(L_i \mid R = r) \leq \frac{E(L)}{\Pr(L_i \geq i)} \qquad (1)$$

As L_i must be at least i provided $R > i$, Equation (1) can be proved as follows:

$$\begin{aligned} E(L_i \mid R = r) &= \sum_{k=i}^{\infty} k \Pr(L_i = k \mid R = r) = \sum_{k=i}^{\infty} k \frac{\Pr(L_i = k \wedge R = r)}{\Pr(R = r)} \\ &= \sum_{k=i}^{\infty} k \frac{\Pr(L_1 \geq 1) \cdots \Pr(L_i = k) \cdots \Pr(L_{r-1} \geq r-1)\Pr(L_r < r)}{\Pr(L_1 \geq 1) \cdots \Pr(L_i \geq i) \cdots \Pr(L_{r-1} \geq r-1)\Pr(L_r < r)} \\ &= \sum_{k=i}^{\infty} k \Pr(L_i = k) \Big/ \Pr(L_i \geq i) \leq \frac{E(L_i)}{\Pr(L_i \geq i)} = \frac{E(L)}{\Pr(L \geq i)} \end{aligned}$$

Similarly, it can be shown that $$E(L_r \mid R = r) \leq E(L)/\Pr(L_r < r) \qquad (2)$$

Furthermore, it is clear that $$E(L_r \mid R = r) \leq r - 1 \qquad (3)$$

Now, we rewrite $E(L_\Sigma)$:

$$E(L_\Sigma) = \underbrace{\sum_{r=1}^{\mu} E(L_\Sigma \mid R = r)\Pr(R = r)}_{(\alpha)} + \underbrace{\sum_{r>\mu} E(L_\Sigma \mid R = r)\Pr(R = r)}_{(\beta)}$$

Using $\Pr(L_i \geq i) \geq 1/2$ for $i \leq \mu$ as well as Equations (1) and (3), we obtain:

$$(\alpha) = \sum_{r=1}^{\mu} \Big(E(L_1 \mid R = r) + \cdots + E(L_r \mid R = r) \Big) \Pr(R = r)$$

$$
\begin{aligned}
&= \sum_{r=1}^{\mu} \left(\sum_{i=1}^{r-1} E(L_i \mid R = r) + E(L_r \mid R = r) \right) \Pr(R = r) \\
&\leq \sum_{r=1}^{\mu} \left(\sum_{i=1}^{r-1} \frac{E(L)}{\Pr(L \geq i)} + (r-1) \right) \Pr(R = r) \\
&\leq \sum_{r=1}^{\mu} ((r-1)2E(L) + (r-1)) \Pr(R = r) \leq (\mu - 1)(2E(L) + 1) = O(\mu\ell)
\end{aligned}
$$

For (β), we use Equations (1) and (2) to obtain:

$$
\begin{aligned}
(\beta) &= \sum_{r=\mu+1}^{\infty} \Big(E(L_1 \mid R = r) + \cdots + E(L_r \mid R = r) \Big) \Pr(R = r) \\
&\leq \sum_{r=\mu+1}^{\infty} \Big(\frac{E(L)}{\Pr(L_1 \geq 1)} + \cdots + \frac{E(L)}{\Pr(L_{r-1} \geq r-1)} + \frac{E(L)}{\Pr(L_r < r)} \Big) \Pr(R = r) \\
&\leq \sum_{r=\mu+1}^{\infty} \Big(\frac{(r-1)E(L)}{\Pr(L_{r-1} \geq r-1)} + \frac{E(L)}{\Pr(L_r < r)} \Big) \Big(\prod_{i=1}^{r-1} \Pr(L_i \geq i) \Big) \Pr(L_r < r) \\
&\leq \sum_{r=\mu+1}^{\infty} rE(L)\Pr(L_1 \geq 1) \cdots \Pr(L_{r-2} \geq r-2) \\
&\leq \sum_{r=\mu+1}^{\infty} rE(L) \left(\frac{1}{2}\right)^{r-2-\mu} \qquad \text{(using } \Pr(L_i \geq i) \leq \tfrac{1}{2} \text{ for } i > \mu) \\
&= E(L) \sum_{r=0}^{\infty} (r + \mu + 1) \left(\frac{1}{2}\right)^{r-1} \\
&= E(L) \Big(\underbrace{\sum_{r=0}^{\infty} r \left(\frac{1}{2}\right)^{r-1}}_{=4} + \underbrace{\sum_{r=0}^{\infty} (\mu+1) \left(\frac{1}{2}\right)^{r-1}}_{=4(\mu+1)} \Big) = E(L)(4\mu + 8) = O(\mu\ell)
\end{aligned}
$$

□

Now, under the same assumptions as in Lemma 9, we can estimate the expected number of steps performed in Phase 2 as follows.

Lemma 10. *During Phase 2, the expected number of steps performed by Algorithm 1LA is $O(\ell)$.*

Finally, we deal with Phase 3. Again, let L be as in Lemma 9. Then, the average amount of time spent in Phase 3 can be estimated as follows.

Lemma 11. *During Phase 3, the expected number of steps performed in calls to the functions first_cand and next_cand is $O(\mu^2 \log \mu)$.*

Lemma 12. *During Phase 3, the expected number of steps performed in reading strings is $O(\mu\ell \log \mu)$.*

Proof. Denote by W_3^r the number of steps performed while reading strings in Phase 3. We make a distinction between strings read before the correct set of candidate patterns is considered, and strings read afterwards until the end of Phase 3. The former are accounted for by random variable V_1, the latter by V_2.

If the correct set of candidate patterns, i.e., the set containing π, is not yet considered, the probability that a new string does not force the correct set of candidate patterns to be considered is at most $1/2$. Denote by K the random variable whose value is the number of strings that are read in Phase 3 before the correct set of candidate patterns is considered. We have:

$$E(V_1) = \sum_{k=0}^{\infty} E(V_1 \mid K = k)\Pr(K = k) \leq \sum_{k=0}^{\infty} kE(L)\left(\frac{1}{2}\right)^k = O(E(L))$$

Assume that the correct set of candidate patterns P' contains M patterns that are considered before pattern π. For any such pattern τ, the probability that a string drawn from $L(\pi)$ according to a proper probability distribution is in the language of τ is at most $1/2$, because either τ has an additional variable or τ has an additional constant symbol (Lemma 8). Denote by V_2^i the steps performed for reading strings while the i-th pattern in P' is considered.

$$E(V_2 \mid M = m) = \sum_{i=1}^{m} E(V_2^i|M = m) \leq \sum_{i=1}^{m}\sum_{k=0}^{\infty} kE(L)\left(\frac{1}{2}\right)^k = O(mE(L))$$

Since $M = O(R \log R)$, we obtain:

$$\begin{aligned}
E(V_2) &= \sum_{r=1}^{\infty} E(V_2 \mid R = r)\Pr(R = r)\\
&= \underbrace{\sum_{r=1}^{\mu} E(V_2 \mid R = r)\Pr(R = r)}_{=O(E(L)\mu\log\mu)} + \underbrace{\sum_{r=\mu+1}^{\infty} E(V_2 \mid R = r)\Pr(R = r)}_{(\alpha)}
\end{aligned}$$

and

$$\begin{aligned}
(\alpha) &\leq \sum_{r=\mu+1}^{\infty} O(E(L) r\log r)\left(\frac{1}{2}\right)^{r-\mu} = E(L)\sum_{r=1}^{\infty} O((r+\mu)\log(r+\mu))\left(\frac{1}{2}\right)^r\\
&= O(\mu E(L)\log\mu)
\end{aligned}$$

Hence, we have $E(W_3^r) = E(V_1) + E(V_2) = O(\mu\ell\log\mu)$. □

Lemma 13. *During Phase 3, the expected number of steps performed in checking whether the current candidate pattern generates a newly read sample string is $O(\mu\ell\log\mu)$.*

Putting it all together, we arrive at the following expected total learning time required by Algorithm 1LA.

Theorem 5. *If the sample strings are drawn from $L(\pi)$ according to a proper probability distribution with expected string length ℓ the expected total learning time of Algorithm 1LA is $O(\ell^2 \log \ell)$.*

5. Learning with Superset Queries

PAT is not learnable with polynomially many queries if only equivalence, membership, and subset queries are allowed provided range$(\mathcal{H}) = PAT$ (cf. [3]). This result may be easily extended to PAT_1. However, positive results are also known. First, *PAT* is exactly learnable using polynomially many disjointness queries with respect to the hypothesis space $PAT \cup FIN$, where *FIN* is the set of all finite languages (cf. [13]). The proof technique easily extends to PAT_1, too. Second, Angluin [3] established an algorithm exactly learning *PAT* with respect to *PAT* by asking $O(|\pi|^2 + |\pi||\mathcal{A}|)$ many *superset queries*. However, it requires choosing *general* patterns τ for asking the queries, and does definitely not work if the *hypothesis space* is PAT_1. Hence, it is natural to ask:

Does there exist a superset query algorithm learning PAT_1 with respect to PAT_1 that uses only polynomially many superset queries?

Using the results of previous sections, we are able to answer this question affirmatively. Nevertheless, whereas *PAT* can be learned with respect to *PAT* by *restricted* superset queries, i.e., superset queries not returning counterexamples, our query algorithm needs counterexamples. Interestingly, it does not need a counterexample for every query answered negatively, instead *two* counterexamples always suffice. The next theorem shows that one-variable patterns are not learnable by a polynomial number of restricted superset queries.

Theorem 6. *Any algorithm exactly identifying all $L \in PAT_1$ generated by a pattern π of length n with respect to PAT_1 by using only restricted superset queries and restricted equivalence queries must make at least $|\mathcal{A}|^{n-2} \geq 2^{n-2}$ queries in the worst case.*

Furthermore, we can show that learning PAT_1 with a polynomial number of superset queries is impossible if the algorithm may ask for a single counterexample only.

Theorem 7. *Any algorithm that exactly identifies all one-variable pattern languages by restricted superset queries and one unrestricted superset query needs at least $2^{(k-1)/4} - 1$ queries in the worst case, where k is the length of the counterexample returned.*

The new algorithm QL works as follows (see Figure 2). Assume the algorithm should learn some pattern π. First QL asks whether $L(\pi) \subseteq L(0) = \{0\}$ holds. This is the case iff $\pi = 0$, and if the answer is *yes* QL knows the right result. Otherwise, QL obtains a counterexample $C(0) \in L(\pi)$. By asking $L(\pi) \subseteq L(C(0)^{\leq j}x)$ for $j = 1, 2, 3, \ldots$ until the answer is *no*, QL computes $i = \min\{\, j \mid L(\pi) \nsubseteq L(C(0)^{\leq j}x)\,\}$. Now we know that π starts with $C(0)^{\leq i-1}$, but what about the i-th position of π? If $|\pi| = i - 1$, then $\pi \in \mathcal{A}^+$ and therefore $\pi = C(0)$. QL asks $L(\pi) \subseteq L(C(0))$ to determine if this is the case. If $L(\pi) \nsubseteq L(C(0))$, then $\#_x(\pi) \geq 1$ and $\pi(i) \notin \mathcal{A}$, since this would imply that $\pi(i) = C(0)(i)$ and, thus, $L(\pi) \subseteq L(C(0)^{\leq i}x)$, a contradiction. Hence, $\pi(i) = x$. Now QL uses the counterexample for the query $L(\pi) \subseteq L(C(0)^{\leq i}x)$ to construct a set $S = \{C(0), C(C(0)^{\leq i}x)\}$. By construction, the two counterexamples differ in their i-th position, but coincide in their first $i - 1$ positions.

```
if L(π) ⊆ L(0) then τ ← 0
else i ← 1;
    while L(π) ⊆ L(C(0)^{≤i} x) do i ← i + 1 od;
    if L(π) ⊆ L(C(0)) then τ ← C(0)
    else S ← {C(0), C(C(0)^{≤i} x)}; R ← Cons(S);
        repeat τ ← max(R); R ← R \ {τ} until L(π) ⊆ L(τ)
    fi
fi; return τ
```

Fig. 2. Algorithm QL. This algorithm learns a pattern π by superset queries. The queries have the form "$L(\pi) \subseteq L(\tau)$," where $\tau \in Pat_1$ is chosen by the algorithm. If the answer to a query $L(\pi) \subseteq L(\tau)$ is *no*, the algorithm can ask for a counterexample $C(\tau)$. By $w^{\leq i}$ we denote the prefix of w of length i and by $\max(R)$ some maximum–length element of R.

Algorithm 2 in Section 2 computes $R = Cons(S)$. Since $S \subseteq L(\pi)$ and since π coincides with S in the first $i-1$ positions, $\pi \in R$. Again we narrowed the search for π to a set R of candidates. Let m be the length of the shortest counterexample in S. Then $|R| = O(m \log m)$ by Lemma 5. Now, the only task left is to find π among all patterns in R. We find π by removing other patterns from R by using the following lemma.

Lemma 14. *Let $S \subseteq \mathcal{A}^+$ and π, $\tau \in Cons(S)$, $|\pi| \leq |\tau|$. Then $L(\pi) \subseteq L(\tau)$ implies $\pi = \tau$.*

QL tests $L(\pi) \subseteq L(\tau)$ for a maximum length pattern $\tau \in R$ and removes τ from R if $L(\pi) \not\subseteq L(\tau)$. Iterating this process finally yields the longest pattern τ for which $L(\pi) \subseteq L(\tau)$. Lemma 14 guarantees $\tau = \pi$. Thus, we have:

Theorem 8. *Algorithm QL learns PAT_1 with respect to PAT_1 by asking only superset queries. The query complexity of QL is $O(|\pi| + m \log m)$ many restricted superset queries plus two superset queries (these are the first two queries answered no) for every language $L(\pi) \in PAT_1$, where m is the length of the shortest counterexample returned.*

Thus, the query complexity $O(|\pi| + m \log m)$ of our learner compares well with that of Angluin's [3] learner (which is $O(|\pi||\mathcal{A}|)$ when restricted to learn PAT_1) using the much more powerful hypothesis space PAT as long as the length of the shortest counterexample returned is not too large.

Acknowledgements

A substantial part of this work has been done while the second author was visiting Kyushu University. This visit has been supported by the Japanese Society for the Promotion of Science under Grant No. 106011.

The fifth author kindly acknowledges the support by the Grant-in-Aid for Scientific Research (C) from the Japan Ministry of Education, Science, Sports, and Culture under Grant No. 07680403.

References

1. D. Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21:46–62, 1980.
2. D. Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45:117–135, 1980.
3. D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.
4. T. Erlebach, P. Rossmanith, H. Stadtherr, A. Steger, and T. Zeugmann. Efficient Learning of One-Variable Pattern Languages from Positive Data. DOI-TR-128, Department of Informatics, Kyushu University, Dec. 12, 1996; http://www.i.kyushu-u.ac.jp/~thomas/treport.html.
5. G. Filé. The relation of two patterns with comparable languages. In *Proc. 5th Ann. Symp. Theoretical Aspects of Computer Science*, LNCS 294, pp. 184–192, Berlin, 1988. Springer-Verlag.
6. S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Ann. ACM Symp. Theory of Computing*, pp. 114–118, New York, 1978, ACM Press.
7. E. M. Gold. Language identification in the limit. *Inf. Control*, 10:447–474, 1967.
8. J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley, 1992.
9. K. P. Jantke. Polynomial time inference of general pattern languages. In *Proc. Symp. Theoretical Aspects of Computer Science*, LNCS 166, pp. 314–325, Berlin, 1984. Springer-Verlag.
10. T. Jiang, A. Salomaa, K. Salomaa, and S. Yu. Inclusion is undecidable for pattern languages. In *Proc. 20th Int. Colloquium on Automata, Languages and Programming,* LNCS 700, pp. 301–312, Berlin, 1993. Springer-Verlag.
11. M. Kearns and L. Pitt. A polynomial-time algorithm for learning k–variable pattern languages from examples. In *Proc. 2nd Ann. ACM Workshop on Computational Learning Theory*, pp. 57–71, Morgan Kaufmann Publ., San Mateo, 1989.
12. K.-I. Ko and C.-M. Hua. A note on the two-variable pattern-finding problem. *J. Comp. Syst. Sci.*, 34:75–86, 1987.
13. S. Lange and R. Wiehagen. Polynomial-time inference of arbitrary pattern languages. *New Generation Computing*, 8:361–370, 1991.
14. S. Lange and T. Zeugmann. Set-driven and rearrangement-independent learning of recursive languages. *Mathematical Systems Theory*, 29:599–634, 1996.
15. D. Osherson, M. Stob and S. Weinstein. *Systems that learn: An introduction to learning theory for cognitive and computer scientists.* MIT Press, 1986
16. L. Pitt. Inductive inference, DFAs and computational complexity. In *Proc. Analogical and Inductive Inference*, LNAI 397, pp. 18–44, Berlin, 1989, Springer-Verlag.
17. A. Salomaa. Patterns. EATCS Bulletin 54:46–62, 1994.
18. A. Salomaa. Return to patterns. EATCS Bulletin 55:144–157, 1994.
19. T. Shinohara and S. Arikawa. Pattern inference. In *Algorithmic Learning for Knowledge-Based Systems*, LNAI 961, pp. 259–291, Berlin, 1995. Springer-Verlag.
20. K. Wexler and P. Culicover. *Formal Principles of Language Acquisition.* MIT Press, Cambridge, MA, 1980.
21. T. Zeugmann. Lange and Wiehagen's pattern language learning algorithm: An average-case analysis with respect to its total learning time. *Annals of Mathematics and Artificial Intelligence*, 1997, to appear.
22. T. Zeugmann and S. Lange. A guided tour across the boundaries of learning recursive languages. In *Algorithmic Learning for Knowledge-Based Systems*, LNAI 961, pp. 190–258, Berlin, 1995. Springer-Verlag.